Contents

Chapter 1

Automatic Diversity for WebAssembly

We aim to create artificial software diversity for WebAssembly, by providing tools to make the process easier and feasible for developers and researchers. As far as we know, there is no software that provides Artificial Software Diversification for WebAssembly. Therefore, we need to enunciate the engineering foundation to implement the strategies defined in **??**. Our implementations are part of the contributions of this thesis. Concretely, we provide two software artifacts that complement this work. Our approach generate WebAssembly program variants statically at compile time to provide randomization. Besides, it provides the tooling to generate MVE binaries for WebAssembly. In this chapter we describe our technical contributions. In Section 1.1 we enunciate how the current state-of-the-art lead us to contribute with Software Diversification through LLVM. We follow by describing our two contributions and their main technical insights in Section 1.2 and Section 1.3. Besides, we describe a new transformation strategy as part of our contributions.

1.1 Global approach

The work of Hilbig et al. [**?**] at 2021 influences our design decisions. According to their work, 70% of the WebAssembly binaries in the wild are created with LLVM-based compilers. Therefore, we decided to provide Artificial Sotfware Diversity for WebAssembly through LLVM. Other solutions would have been to diversify at the source code level, or at the WebAssembly binary level. However, the former would limit the applicability of our work. The latter, is proposed for future works.

LLVM is compound of three main components [**?**]. First, the frontend (compilers such as clang and rustc) converts the program source code to LLVM intermediate representation (LLVM IR). Second, optimization and transformation passes improve the LLVM IR. Third and final, the backend component is in charge

of generating the target machine code. In Figure 1.1 we show how we use the LLVM pipeline in our contributions, which are highlighted as dashed squares.

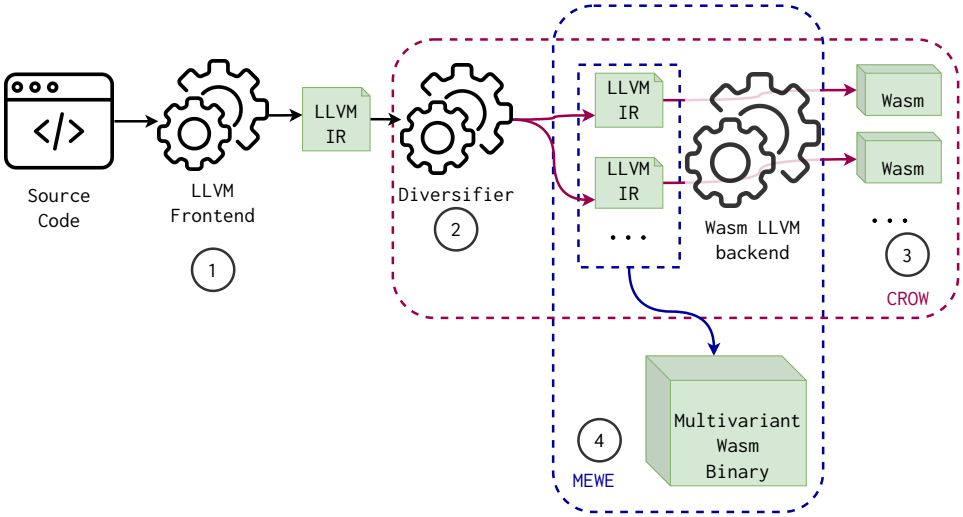

Figure 1.1: Generic workflow to create WebAssembly program variants.

The global workflow in Figure 1.1 starts by receiving source code. Then, the LLVM frontend transforms it into LLVM IR representation (1). We alter the LLVM pipeline that compiles source code to Wasm, with the introduction of a Diversifier component. The diversifier generates LLVM IR variants from of the output of the frontend (2). The LLVM IR variants are inputs for our customized Wasm backend. The Diversifier and the custom Wasm LLVM backend compose CROW, which creates WebAssembly program variants out of a source code program (3). In addition, an orthogonal tool comes from the generation of LLVM IR variants at Step (2). MEWE [**?**], merges and creates multivariant binaries to provide MVE for WebAssembly (4).

1.2 CROW: Code Randomization Of WebAssembly

This section describes the red squared tooling in Figure 1.1 named, CROW [**?**]. CROW is a tool tailored to create semantically equivalent WebAssembly variants from the output of an LLVM frontend. Using a custom Wasm LLVM backend, it generates the Wasm binary variants.

In Figure 1.2, we describe the workflow of CROW to create program variants. The Diversifier in CROW is composed by two main processes, *exploration* and *combining*. The *exploration* process operates at the instruction level for each function

in its input LLVM. For each instruction, CROW produces a collection of functionally equivalent code replacements. In the *combining* stage, CROW assembles the code replacements to generate different LLVM IR variants. CROW generates the LLVM IR variants by traversing the power set of all possible combinations of code replacements. Finally, the custom Wasm LLVM backend compiles the assembled LLVM IR variants into WebAssembly binaries. In the following, we describe our design decisions. All our implementation choices are based on one premise: *each design decision should increase the number of WebAssembly variants that CROW creates.*

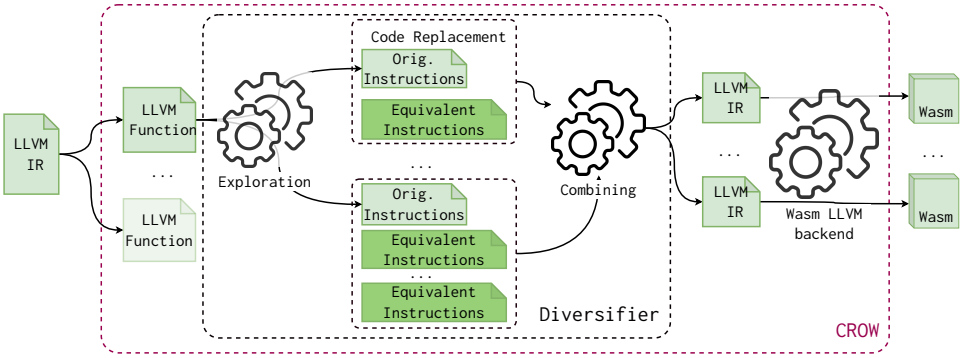

Figure 1.2: CROW components following the diagram in Figure 1.1. CROW takes LLVM IR to generate functionally equivalent code replacements. Then, CROW assembles program variants by combining them.

1.2.1 Exploration

The key component of CROW's exploration process is its code replacements generation strategy. The diversifier implemented in CROW is based on the superdiversifier of Jacob et al. [**?**]. A superoptimizer focuses on *searching* for a new program which is faster or smaller than the original code, while preserving its functionality. The concept of superoptimizing a program dates back to 1987, with the seminal work of Massalin [**?**] which proposes an exhaustive exploration of the solution space. The search space is defined by choosing a subset of the machine's instruction set and generating combinations of optimized programs, sorted by length in ascending order. If any of these programs are found to perform the same function as the source program, the search halts. On the contrary, a superdiversifier keeps all intermediate search results despite their performance.

We use the same idea of Jacob and colleagues to implement CROW because of two main reasons. First, the code replacements generated by this technique

outperform diversification strategies based on hand-written rules. Besides, this technique is fully automatic. Second, there is a battle tested superoptimizer for LLVM, Souper [**?**]. This latter, makes feasible the construction of a generic LLVM superdiversifier.

We modify Souper to keep all possible solutions in their searching algorithm. Souper builds a Data Flow Graph for each LLVM integer-returning instruction. Then, for each Data Flow Graph, Souper exhaustively builds all possible expressions from a subset of the LLVM IR language. Each syntactically correct expression in the search space is semantically checked versus the original with a theorem solver. Souper synthesizes the replacements in increasing size. Thus, the first found equivalent transformation the optimal replacement result of the searching. We keep more equivalent replacements during the searching by removing the halting criteria. Instead, we limit the searching for replacement with timeout and the replacement's size. Our customized Souper reports a new code replacement as soon as an equivalent transformation is found.

Notice that the searching space exponentially increases with the size of the LLVM IR language subset. Thus, we prevent Souper from synthesizing instructions that have no correspondence in the WebAssembly backend. This decision reduces the searching space. For example, creating expression having the `freeze` LLVM instructions will increase the searching space for an instruction without a Wasm's opcode in the end. Moreover, we disable the majority of the pruning strategies of Souper for the sake of more variants.

1.2.2 Constant inferring

One of the code transformation strategies of Souper does *constant inferring*. This means that Souper infers pieces of code as a single constant assignment. In particular, Souper focuses on boolean valued variables that are used to control branches. By extending Souper as a superdiversifier, we add this transformation strategy as a new mutation strategy to the ones defined in **??**.

After a *constant inferring*, the generated program is considerably different to the original program, being good for diversification. Let us illustrate the case with an example. The Babbage problem code in Listing 1.1 is composed of a loop which stops when it discovers the smaller number that fits with the Babbage condition in Line 4.

Listing 1.1: Babbage problem.

```
1     int babbage() {
2         int current = 0,
3             square;
4         while ((square=current*current) % 1000000
                ↪ != 269696) {
5             current++;
6         }
7         printf ("The number is %d\n", current);
8         return 0 ;
9     }
```

Listing 1.2: Constant inferring transformation over the original Babbage problem in Listing 1.1.

```
int babbage() {
    int current = 25264;

    printf ("The number is %d\n", current);
    return 0 ;
}
```

In theory, this value can also be inferred by unrolling the loop the correct number of times with the LLVM toolchain. However, standard LLVM tools cannot unroll the `while`-loop because the loop count is too large. The original Souper deals with this case, generating the program in Listing 1.2. It infers the value of `current` in Line 2 such that the Babbage condition is reached. Therefore, the condition in the loop will always be false. Then, the loop is dead code, and is removed in the final compilation. It is clear that the new program in Listing 1.2 is remarkably smaller and faster than the original code. Therefore, offers differences both statically and at runtime. [1]

1.2.3 Removing latter optimizations for LLVM

During the implementation of CROW we have the premise of removing all builtin optimizations in the LLVM backend that could reverse Wasm variants. Therefore, in addition to the extended Souper, we modify the WebAssembly backend. We disable all optimizations in the WebAssembly backend that could reverse the superoptimizer transformations, such as constant folding and instructions normalization.

1.3 MEWE: Multi-variant Execution for WEbAssembly

This section describes MEWE [**?**]. MEWE synthesizes diversified function variants by using CROW. It then provides execution-path randomization in a Multivariant Execution (MVE. The tool generates application-level multivariant binaries, without any change to the operating system or WebAssembly runtime. MEWE creates an MVE by intermixing functions for which CROW generates variants, as step ② in Figure 1.1 shows. CROW generates each one of these variants with fine-grained diversification at instruction level, applying the majority

[1] Notice that for the sake of illustration, we show both codes in C language, this process inside CROW is performed directly in LLVM IR. Also notice that the two programs in the example follow the definition of *functional equivalence* discussed in **??**.

of the strategies discussed in **??** and *constant inferring*. MEWE adds a new mutation strategy. It inlines function variants when appropriate, also resulting in call stack diversification at runtime.

In Figure 1.3 we zoom MEWE from the blue highlighted square in Figure 1.1. MEWE takes the LLVM IR variants generated by CROW's diversifier and merges them into a Wasm multivariant. In the figure, we highlight the two components of MEWE, *Multivariant Generation* and the *Mixer*. In the *Multivariant Generation* process, MEWE merges the LLVM IR variants created by CROW, and creates a LLVM multivariant binary. The merging of the variants intermixes the calling of function variants, making possible the execution path randomization. *The Mixer* augments the LLVM multivariant binary with a random generator, which is required for performing the execution-path randomization. Also, *The Mixer* fixes the entrypoint in the multivariant binary. Finally, MEWE generates a standalone multivariant WebAssembly binary using the same custom Wasm LLVM backend from CROW. The generated multivariant WebAssembly binary can be deployed as it is to any engine.

1.3.1 Multivariant generation

The key component of MEWE consists in combining the variants into a single binary. The goal is to support execution-path randomization at runtime. The core idea is to introduce one dispatcher function per original function with variants. A dispatcher function is a synthetic function that is in charge of choosing a variant at random, every time the original function is invoked during the execution. With the introduction of dispatcher function, MEWE turns the original call graph into a multivariant call graph, defined as follows.

Definition 1. *Multivariant Call Graph (MCG): A multivariant call graph is a call graph $\langle N, E \rangle$ where the nodes in N represent all the functions in the binary and an edge $(f_1, f_2) \in E$ represents a possible invocation of f_2 by f_1 [**?**], where the nodes are typed. The nodes in N have three possible types: a function present in the original program, a generated function variant, or a dispatcher function.*

In Figure 1.4, we show the original static call graph for and original program (top of the figure), as well as the multivariant call graph generated with MEWE (bottom of the figure). The grey nodes represent function variants, the green nodes function dispatchers and the yellow nodes are the original functions. The possible calls are represented by the directed edges. The original program includes 3 functions. MEWE generates 43 variants for the first function, none for the second and three for the third function. MEWE introduces two dispatcher nodes, for the first and third functions. Each dispatcher is connected to the corresponding function variants, in order to invoke one variant randomly at runtime.

In Listing 1.3, we illustrate the LLVM construction for the function dispatcher corresponding to the right most green node of Figure 1.4. It first calls the random generator, which returns a value that is then used to invoke a specific function

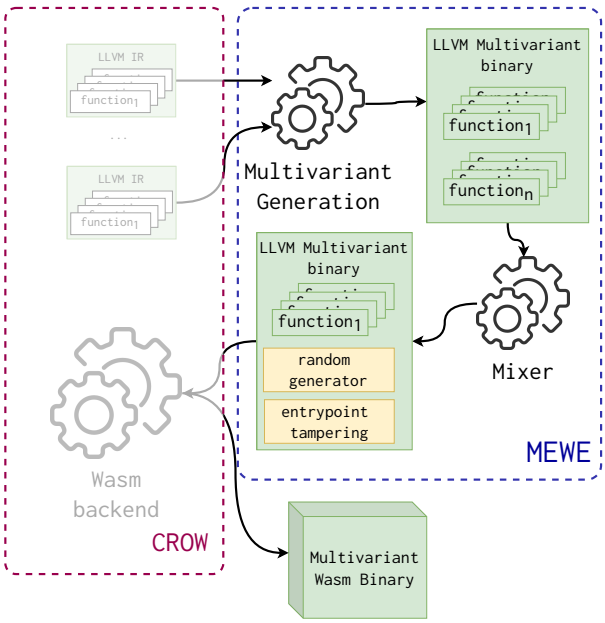

Figure 1.3: Overview of MEWE workflow. It takes as input an LLVM binary. It first generates a set of functionally equivalent variants for each function in the binary using CROW. Then, MEWE generates a LLVM multivariant binary composed of all the function variants. The Mixer includes the behavior in charge of selecting a variant when a function is invoked. The MEWE mixer composes the LLVM multivariant binary with a random number generation library and a tampering of the original application entrypoint. The final process produces a WebAssembly multivariant binary ready to be deployed.

variant. We implement the dispatchers with a switch-case structure to avoid indirect calls that can be susceptible to speculative execution based attacks [**?**]. The choice of a switch-case also avoids having multiple function definitions with the same signature, which could increase the attack surface in case the function signature is vulnerable [**?**]. This also allows MEWE to inline function variants inside the dispatcher, instead of defining them again. Here we trade security over performance, since dispatcher functions that perform indirect calls, instead of a switch-case, could improve the performance of the dispatchers as indirect calls have constant time.

1.3.2 The Mixer

MEWE has four specific objectives: link the LLVM multivariant binary, inject a random generator, tamper the entrypoint of the application, and merge all

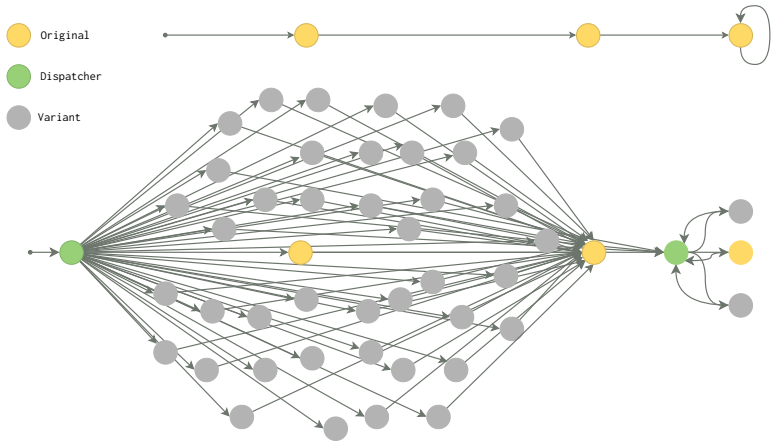

Figure 1.4: Example of two static call graphs. At the top, the original call graph, at the bottom, the multivariant call graph, which includes nodes that represent function variants (in grey), dispatchers (in green), and original functions (in yellow).

```
define internal i32 @foo(i32 %0) {
    entry:
      %1 = call i32 @discriminate(i32 3)
      switch i32 %1, label %end [
        i32 0, label %case_43_
        i32 1, label %case_44_
      ]
    case_43_:
      %2 = call i32 @foo_43_(%0)
      ret i32 %2
    case_44_:
      %3 = <body of foo_44_ inlined>
      ret i32 %3
    end:
      %4 = call i32 @foo_original(%0)
      ret i32 %4
}
```

Listing 1.3: Dispatcher function embedded in the multivariant binary of the original function in the rightmost green node in Figure 1.4.

these components into a multivariant WebAssembly binary. We use the Rustc compiler[2] to orchestrate the mixing. For the random generator, we rely on WASI's specification [**?**] for the random behavior of the dispatchers. Its exact implementation is dependent on the platform on which the binary is deployed. The Mixer creates a new entrypoint for the binary called *entrypoint tampering*. It wraps the dispatcher for the entrypoint variants as a new function for the final Wasm binary and is declared as the application entrypoint.

1.4 Accompanying Source Code

This thesis is accompanied by the source code of both contributions, CROW and MEWE. The source code is accessible through the links:

1. CROW: `https://github.com/KTH/slumps`
2. MEWE: `https://github.com/Jacarte/MEWE`

Our software artifacts are licensed under the MIT License. The dependent source codes, such as LLVM, are licensed under their original licenses.

Conclusions

This chapter discusses the technical details for the tools implemented for our main contributions. We describe how CROW generates program variants for the sake of software diversification. We propose a global architecture for a generic LLVM superdiversifier We introduce a new mutation strategy that is a consequence of retargeting Souper as a superdiversifier. Besides, we dissect MEWE and how it creates an MVE system. In **??** we discuss the methodology we follow to evaluate how CROW and MEWE creates software diversification.

[2] `https://doc.rust-lang.org/rustc/what-is-rustc.html`

Bibliography

[] Narayan,S., Disselkoen,C., Moghimi,D., Cauligi,S., Johnson,E., Gang,Z., Vahldiek-Oberwagner,A., Sahita,R., Shacham,H., Tullsen,D., et al. (2021). Swivel: Hardening webassembly against spectre. In *USENIX Security Symposium*.

[] Johnson,E., Thien,D., Alhessi,Y., Narayan,S., Brown,F., Lerner,S., McMullen,T., Savage,S., and Stefan,D. (2021). Sfi safety for native-compiled wasm. *NDSS. Internet Society*.

[] Hilbig,A., Lehmann,D., and Pradel,M. (2021). An empirical study of real-world webassembly binaries: Security, languages, use cases. *Proceedings of the Web Conference 2021*.

[] Cabrera Arteaga,J., Laperdrix,P., Monperrus,M., and Baudry,B. (2021). Multi-Variant Execution at the Edge. *arXiv e-prints*, page arXiv:2108.08125.

[] Cabrera Arteaga,J., Floros,O., Vera Perez,O., Baudry,B., and Monperrus,M. (2021). Crow: code diversification for webassembly. In *MADWeb, NDSS 2021*.

[] (2021). Webassembly system interface.

[] Sasnauskas,R., Chen,Y., Collingbourne,P., Ketema,J., Lup,G., Taneja,J., and Regehr,J. (2017). Souper: A Synthesizing Superoptimizer. *arXiv preprint 1711.04422*.

[] Jacob,M., Jakubowski,M. H., Naldurg,P., Saw,C. W. N., and Venkatesan,R. (2008). The superdiversifier: Peephole individualization for software protection. In *International Workshop on Security*, pages 100–120. Springer.

[] LLVM (2003). The LLVM Compiler Infrastructure .

[] Henry,M. (1987). Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126.

[] Ryder,B. G. (1979). Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226.